Annexe

Noise-contrastive estimation of normalising constants and GANs

Ce document est une annexe de notre mémoire constituant une documentation de notre code et reprenant les résultats numériques principaux.

Nous utilisons directement les notations déjà présentées dans le manuscrit du mémoire.

Utilisation : cliquer sur RUN ALL dans Rstudio pour faire tourner les algorithmes à partir des données pré-générées. Celles-ci sont enregistrées sous forme de fichiers .csv dans le dossier **dataframes** du github. Pour générer de nouvelles données, retirer les guillemets dans les cellules de code concernées.

Contents

1 Cadre de travail R

```
#------------- CADRE DE TRAVAIL RSTUDIO ----------------------------------------

# Packages utilisés dans le code

library(ggplot2)
library(reshape2)
library(matrixStats)
library(knitr)
library(moments)
library(tseries)
library(readr)
library(Rcpp)
library(GiRaF)


# Palettes pour les graphiques

pal5 = c("#3B9AB2", "#78B7C5", "#EBCC2A", "#DC863B", "#E1AF00")
pal4 = c("#3B9AB2", "#78B7C5", "#EBCC2A", "#DC863B")
pal3 = c("#3B9AB2", "#EBCC2A", "#DC863B")
pal2 = c("#3B9AB2", "#DC863B")
pal2_bis = c("#3B9AB2", "#EBCC2A")
pal1 = "#3B9AB2"
```

2 Fonctions utilitaires

2.1 Algorithme d'Hasting

Utilité : simuler selon $p_m(.,\psi) := \frac{\overline{p_m}(.,\psi)}{Z(\psi)}$ pour un paramètre ψ choisi, sans connaissance de la constante de normalisation $Z(\psi)$.

Entrée	**Type**	**Exemple**	**Indication**
x	vecteur	rnorm(100, 2, 4)	notre échantillon de densité inconnue
n	entier	100	taille de l'échantillon simulé
h	fonction		fonction qui retourne $\overline{p_m}(.,\psi)$
psi	vecteur	c(mean(x),sd(x))	paramètres de la fonction h
Sortie	**Type**	**Exemple**	**Indication**
y	vecteur		notre échantillon simulé

```
hasting = function(x, n, psi, h){
  # on tire le premier état de la chaine de markov dans l'échantillon de données initiales
  y = c()
  y = append(y, sample(x, 1))
  for (i in 2:n){
    # on tire une proposition pour le nouvel état de la chaîne
    y_ = rnorm(1, y[i-1], 1)
    # on calcule la probabilité d'acceptation
    u = runif(1)
    if ( u <= (h(y_,psi) * dnorm(y_, y[i-1], 1)) / (h(y[i-1],psi) * dnorm(y[i-1], y_, 1))){
    # le nouvel état est accepté
      y = append(y, y_)}
    else {
    # le nouvel état n'est pas accepté
      y = append(y, y[i-1])
    }
  }
  return (y)
}
```

2.2 Algorithme d'Hasting i.i.d.

Utilité : imiter la simulation d'un échantillon de données i.i.d. à partir de l'algorithme d'Hasting

Argument	**Type**	**Exemple**	**Indication**
x	vecteur	rnorm(100, 2, 4)	notre échantillon de densité inconnue
n	entier	100	taille de l'échantillon simulé
h	fonction		fonction qui retourne $\overline{p_m}(., \psi)$
psi	vecteur	c(mean(x),sd(x))	paramètres de la fonction h
eps	entier	100	pas de décorrélation
Sortie	**Type**	**Exemple**	**Indication**
y	vecteur		notre échantillon simulé

```
hasting_iid = function(x, n, psi, h, eps){
  # on tire le premier état de la chaine de markov dans l'échantillon de données initiales
  y = c()
  y = append(y, sample(x, 1))
  for (i in 2:(n*eps)){
    # on tire une proposition pour le nouvel état de la chaîne
    y_ = rnorm(1, y[i-1], 1)
    # on calcule la probabilité d'acceptation
    u = runif(1)
    if ( u <= (h(y_,psi) * dnorm(y_, y[i-1], 1))/(h(y[i-1],psi) * dnorm(y[i-1], y_, 1))){
    # le nouvel état est accepté
      y = append(y, y_)}
    else {
    # le nouvel état n'est pas accepté
      y = append(y, y[i-1])
    }
  }
  # on ne conserve qu'un état tous les "eps" pas
  filter = y * rep(c(1,rep(0, eps-1)), n)
  return (filter[filter != 0])
}
```

3 Méthodes d'estimation des paramètres et de la constante

Les fonctions présentées ci dessous sont valables en dimension de taille 1, c'est à dire lorsqu'un point de donnée est un scalaire et que l'échantillon de données forme un vecteur. Elles seront à adapter pour des dimensions plus élevées, par exemple dans notre application au modèle d'Ising.

3.1 Monte Carlo Maximum Likelihood Estimation (MC MLE)

Utilité : retourne une estimation de α^* selon la méthode MC MLE.

Entrée	**Type**	**Exemple**	**Indication**
x	vecteur	rnorm(100, 2, 4)	notre échantillon de densité inconnue
n	entier	100	taille de l'échantillon de bruit
h	fonction		fonction qui retourne $\overline{p_m}(., \psi)$ (bruit)
psi	vecteur	c(mean(x),sd(x))	paramètres de la fonction h
Sortie	**Type**	**Exemple**	**Indication**
$\widehat{\alpha^*}$	vecteur		estimation de α^*

```
mc_mle = function(x, n, psi, h){
  m = length(x)

  # on génère l'échantillon de bruit
  y = hasting(x, n, psi, h)

  # on pose la fonction objectif
  L = function(alpha){
    return(sum(log(h(x,alpha)/h(x,psi))) - m*log(mean(h(y,alpha)/h(y,psi))))
  }

  # on estime alpha par l'algorithme du gradient conjugué
  alpha = optim(
    par = rep(1,length(psi)),
    gr = "CG",
    control = list(fnscale=-1),
    fn = L
  )$par

  return(alpha)
}
```

3.2 Régression logistique inverse naïve

Utilité : retourne une estimation de $Z(\alpha^*)$ en utilisant pour le bruit la loi p_m pour un paramètre ψ choisi.

Entrée	**Type**	**Exemple**	**Indication**
x	vecteur	rnorm(100, 2, 4)	notre échantillon de densité inconnue
alpha	vecteur	c(2,4)	paramètres de h pour le modèle
n	entier	100	taille de l'échantillon de bruit
psi	vecteur	c(mean(x),sd(x))	paramètres de h pour le bruit
h	fonction		fonction qui retourne $\overline{p_m}(.,.)$
eps	entier	100	pas de décorrélation
Sortie	**Type**	**Exemple**	**Indication**
$\widehat{Z(\alpha^*)}$	nombre		estimation de la constante de normalisation à une constante près

```
rev_log_reg_naive = function(x, alpha, n, psi, h, eps){

  m = length(x)
  y = hasting_iid(x, n, psi, h, eps)

  # calcul des probabilités p_X et p_Y
  denom = function(sample, eta) {
    return(pm_barre(sample, alpha)*exp(eta[1]) + pm_barre(sample,psi)*exp(eta[2]))}
  p_X = function(sample, eta){
    return (pm_barre(sample, alpha)*exp(eta[1]) / denom(sample, eta))}
  p_Y = function(sample, eta){
    return (pm_barre(sample, psi)*exp(eta[2]) / denom(sample, eta))}

  # fonction objectif
  L = function(eta) {
    return(sum(log(p_X(x, eta))) + sum(log(p_Y(y, eta))))}

  # initialisation descente de gradient
  eta1 = -log(sd(x)*sqrt(mean(x)*pi)) + log(m/(m+n))
  eta2 = -log(sd(y)*sqrt(mean(y)*pi)) + log(n/(m+n))

  # optimisation
  eta = optim(
    par = c(eta1,eta2),
    gr = "CG",
    control = list(fnscale=-1),
    fn = L
  )$par

  const = exp(-eta[1] + log(m/(m+n)))
  return(const)
}
```

3.3 Régression logistique inverse

Utilité : retourne une estimation de $Z(\alpha^*)$ en utilisant pour le bruit une loi dont on connait la constante de normalisation.

Entrée	**Type**	**Exemple**	**Indication**
x	vecteur	rnorm(100, 2, 4)	notre échantillon de densité inconnue
generate_y	fonction	rcauchy	générateur de la loi p_{noise} de paramètres n et psi
n	entier	100	taille de l'échantillon de bruit
alpha	vecteur	c(2,4)	paramètres de la fonction h1 (modèle)
psi	vecteur	c(mean(x),sd(x))	paramètres des fonctions h2 et generate_y (bruit)
h1	fonction		fonction qui retourne $\overline{p_m}(.,\alpha)$ (modèle)
h2	fonction		fonction qui retourne $\overline{p_{noise}}(.,\psi)$ (bruit)
Sortie	**Type**	**Exemple**	**Indication**
$\widehat{Z(\alpha^*)}$	nombre		estimation de la constante de normalisation

```
rev_log_reg = function(x, generate_y, n, alpha, psi, h1, h2){

  y = do.call(generate_y, c(list(n),psi))
  m = length(x)

  # calcul des probabilités p_X et p_Y
  denom = function(sample, eta) {
    return(h1(sample, alpha)*exp(eta[1]) + h2(sample,psi)*exp(eta[2]))}
  p_X = function(sample, eta){
    return (h1(sample, alpha)*exp(eta[1]) / denom(sample, eta))}
  p_Y = function(sample, eta){
    return (h2(sample, psi)*exp(eta[2]) / denom(sample, eta))}

  # fonction objectif
  L = function(eta) {
    return(sum(log(p_X(x, eta))) + sum(log(p_Y(y, eta))))}

  # initialisation descente de gradient
  eta1 = log(m/(m+n))
  eta2 = log(n/(m+n))

  # optimisation
  eta = optim(
    par = c(eta1,eta2),
    gr = "CG",
    control = list(fnscale=-1),
    fn = L
  )$par

  constante_additive = eta[2] + log(1) - log(n/(n+m))
  const1 = exp(-eta[1] + log(m/(m+n)) + constante_additive)
  const2 = exp(-eta[2] + log(n/(m+n)) + constante_additive)

  return(const1)
}
```

3.4 Noise Constrastive Estimation

Utilité : Retourne l'estimation du paramètre α^* et de la constante de normalisation $Z(\alpha^*)$ par la méthode NCE.

Argument	**Type**	**Exemple**	**Indication**
x	vecteur	rnorm(100, 2, 4)	notre échantillon de densité inconnue
generate_y	fonction	rcauchy	fonction qui retourne un échantillon suivant la loi p_n
psi	vecteur	c(mean(x),sd(x))	arguments de la fonction generate_y
log_pm	fonction		fonction qui retourne le logarithme de la densité p_m
log_pn	fonction		fonction qui retourne le logarithme de la densité p_n
size_theta	entier	3	taille de $\theta := (\alpha, c)$, vaut généralement 2 ou 3
n	entier	100	taille de l'échantillon de bruit suivant la loi p_n
Sortie	**Type**	**Exemple**	**Indication**
$\widehat{\theta^*} := (\widehat{\alpha^*}, \widehat{Z(\alpha^*)})$	vecteur		estimation du paramètre et de la constante

```
nce = function(x, generate_y, psi, log_pm, log_pn, size_theta, n){
  m = length(x)

  # on simule l'échantillon de bruit
  y = do.call(generate_y, c(list(n),psi))

  # on pose la fonction objectif
  h = function(u, theta){
    return( 1 / (1 + n/m * exp(log_pn(u, psi) - log_pm(u, theta))))
  }

  J = function(theta){
    return( sum(log(h(x, theta))) + sum(log(1 - h(y, theta))) )
  }

  # on estime theta par l'algorithme du gradient conjugué
  theta = optim(
    par = rep(1, size_theta),
    gr = "CG",
    control = list(fnscale=-1),
    fn = J
  )$par

  # on calcule l'estimation de la constante de normalisation à partir de l'estimation de c
  const = exp(-theta[size_theta])
  alpha = theta[-size_theta]

  return(c(alpha, const))
}
```

4 Illustration des méthodes : la loi normale

4.1 Application des méthodes

Dans cette section on applique les méthodes présentées ci-dessus à l'exemple conducteur de du mémoire, la loi normale $\mathcal{N}(2, 4)$.

```
#------------ DENSITES NORMALISEES ET NON NORMALISEES NECESSAIRES DANS LES METHODES

pm_barre = function(u, theta){
  # theta[1] = mu
  # theta[2] = sigma
  return(exp(-0.5 * ((u - theta[1]) / theta[2]) ** 2))
}

log_pm = function(u,theta){
  # theta[1] = mu
  # theta[2] = sigma
  # theta[3] = c
  return(theta[3] - 1/2 * (u/theta[2] - theta[1]/theta[2]) ** 2)
}

pn_cauchy = function(u, psi){
  return(dcauchy(u, psi[1], psi[2]))
}

pn_norm = function(u, psi){
  return(dnorm(u, psi[1], psi[2]))
}

pn_unif = function(u, psi){
  return(dunif(u, psi[1], psi[2]))
}

log_pn_cauchy = function(u,psi){
  return(log(dcauchy(u, psi[1], psi[2])))
}

log_pn_unif = function(u,psi){
  return(log(dcauchy(u, psi[1], psi[2])))
}

#------------ CHOIX DES DIMENSIONS ET DES PARAMETRES --------------------------

m = 10000
n = 100000
x = rnorm(m, 2, 4)
alpha = c(2,4)
psi = c(mean(x), sd(x))
psi2 = c(min(x), max(x))
size_theta = 3

#------------ METHODE MC MLE --------------------------------------------------
mc_mle(x, n, psi, pm_barre)
```

```
## [1] 2.127430 4.056329
```

```
#------------- METHODE REGRESSION LOGISTIQUE INVERSE ----------------------------
rev_log_reg(x, rcauchy, n, alpha, psi, pm_barre, pn_cauchy)
```

```
## [1] 10.04484
```

```
#------------- METHODE NOISE CONTRASTIVE ESTIMATION ----------------------------
# METHODE NCE
nce(x, rcauchy, psi, log_pm, log_pn_cauchy, size_theta, n)
```

```
## [1]  2.044874  4.007402 10.046191
```

4.2 Graphiques

Nous allons à présent réaliser plusieurs graphiques pour chaque méthode afin d'étudier l'impact du choix des paramètres et dimensions. Les données sont tirées des fichiers .csv que nous avons préalablement générés. Les fichiers .csv sont disponibles dans le dossier **/dataframes** du github.

Les graphiques obtenus sont expliqués et commentés dans le manuscrit du mémoire.

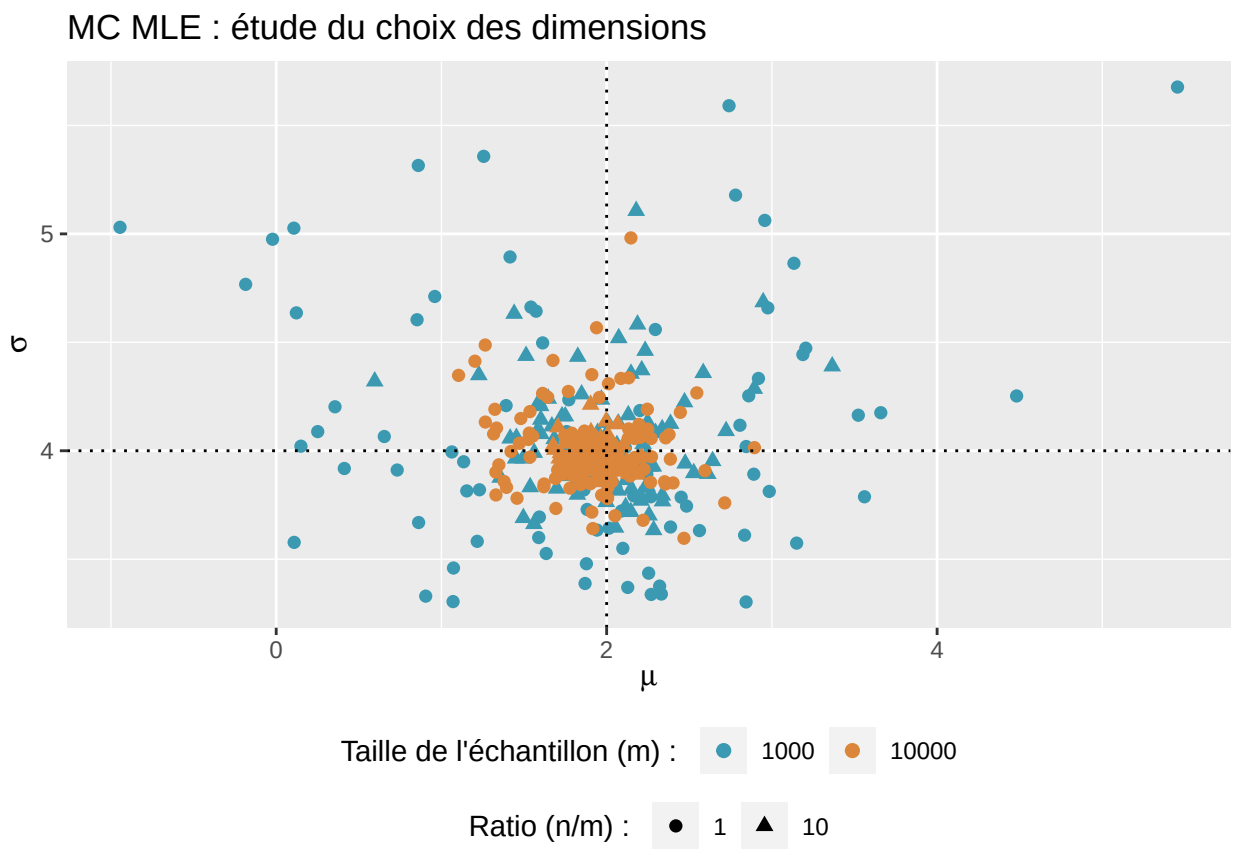
```
'

#------------- CREATION DES CSV POUR LES GRAPHIQUES ----------------------------

RATIO = c(1, 10)
M1 = c(1000, 10000)
M2 = c(1000, 10000, 100000)
M3 = c(1000, 5000, 10000, 15000)

#------------- DF_MCMLE ----------------------------------------------------------

df_mcmle = data.frame(matrix(ncol = 4, nrow = 0))
colnames(df_mcmle) = c("param_1", "param_2", "size_data", "ratio_noise_data")

for (m in M1){

  x_csv = rnorm(m, 2, 4)

  for (r in RATIO) {
    for (i in 1:100) {

       df_mcmle[nrow(df_mcmle) + 1, ] = c(mc_mle(x_csv, m*r, psi, pm_barre), m, r)

    }
  }
}

write.csv(df_mcmle, "df_mcmle.csv", row.names = FALSE)

#------------- DF_MCMLE_PSI ------------------------------------------------------

df_mcmle_psi = data.frame(matrix(ncol = 3, nrow = 0))
colnames(df_mcmle_psi) = c("param_1", "param_2", "ratio_alpha_psi")

for (r in c(1,3,5,7)){
```

```
  for (i in 1:100) {

     df_mcmle_psi[nrow(df_mcmle_psi) + 1, ] = c(mc_mle(x, n, c(2*r, 4*r), pm_barre), r)

  }
}

write.csv(df_mcmle_psi, "dataframes/df_mcmle_psi.csv", row.names = FALSE)

#------------- DF_NCE ---------------------------------------------------------

df_nce = data.frame(matrix(ncol = 5, nrow = 0))
colnames(df_nce) = c("param_1", "param_2", "const", "size_data", "ratio_noise_data")

for (m in M2){

  x_csv = rnorm(m, 2, 4)

  for (r in RATIO) {
    for (i in 1:50) {

       df_nce[nrow(df_nce) + 1, ] = c(nce(x_csv, rcauchy, psi, log_pm, log_pn_cauchy, size_theta, m*r),

    }
  }
}
write.csv(df_nce, "dataframes/df_nce.csv", row.names = FALSE)

#------------- DF_NCE_NOISES --------------------------------------------------

df_nce_noises = data.frame(matrix(ncol = 4, nrow = 0))
colnames(df_nce_noises) = c("param_1", "param_2", "const", "law")

for (i in 1:100) {

   df_nce_noises[nrow(df_nce_noises) + 1, ] = c(nce(x, runif, psi2, log_pm, log_pn_unif, size_theta, n)

}
for (i in 1:100) {

   df_nce_noises[nrow(df_nce_noises) + 1, ] = c(nce(x, rcauchy, psi, log_pm, log_pn_cauchy, size_theta,

}

write.csv(df_nce_noises, "dataframes/df_nce_noises.csv", row.names = FALSE)

#------------- DF_REV_LOG_REG_NAIVE-------------------------------------------

df_rev_log_reg_naive = data.frame(matrix(ncol = 4, nrow = 0))
colnames(df_rev_log_reg_naive) = c("const", "size_data", "ratio_noise_data", "ratio_alpha_psi")

for (m in M3){
```

```
  x = rnorm(m, 2, 4)

  for (r in RATIO) {
    for (coef in 1:4) {
      for (i in 1:15) {

         df_rev_log_reg_naive[nrow(df_rev_log_reg_naive) + 1, ] = c(rev_log_reg_naive(x, alpha, r*m, al

      }
    }
  }
}

write.csv(df_rev_log_reg_naive, "dataframes/df_rev_log_reg_naive.csv", row.names = FALSE)

#------------ DF_REV_LOG_REG ------------------------------------------------

df_rev_log_reg = data.frame(matrix(ncol = 4, nrow = 0))
colnames(df_rev_log_reg) = c("const", "size_data", "ratio_noise_data", "law_noise")

for (m in M3){

  x = rnorm(m, 2, 4)

  for (r in RATIO) {
    for (i in 1:1000) {

       df_rev_log_reg[nrow(df_rev_log_reg) + 1, ] = c(rev_log_reg(x, rcauchy, r*m, alpha, alpha, pm_bar

    }
    for (i in 1:1000) {

       df_rev_log_reg[nrow(df_rev_log_reg) + 1, ] = c(rev_log_reg(x, rnorm, r*m, alpha, alpha, pm_barre

    }
    for (i in 1:1000) {

       df_rev_log_reg[nrow(df_rev_log_reg) + 1, ] = c(rev_log_reg(x, runif, r*m, alpha, psi2, pm_barre,

    }
  }
}

write.csv(df_rev_log_reg, "dataframes/df_rev_log_reg.csv", row.names = FALSE)

'
```

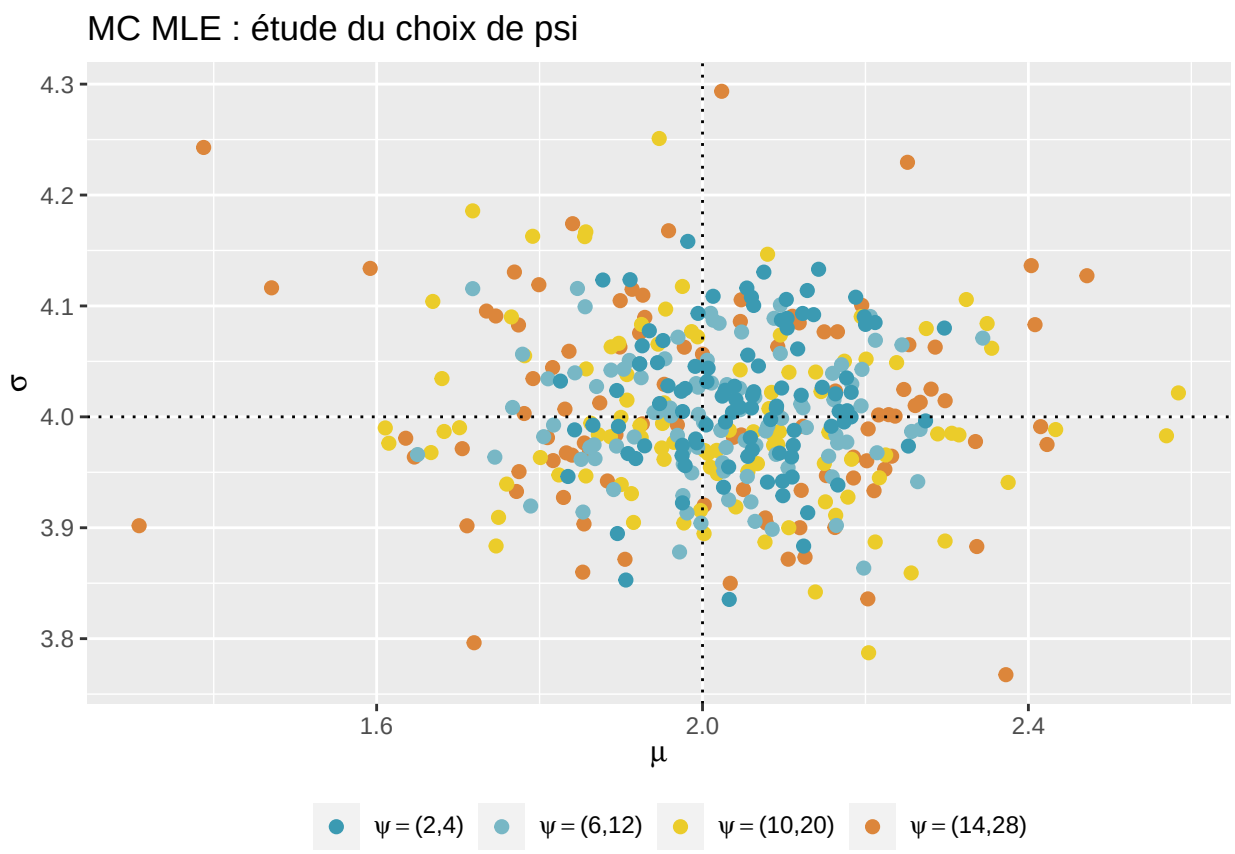
```
#------------ METHODE MC MLE : ETUDE DES DIMENSIONS ---------------------------

df_mcmle <- read_csv("dataframes/df_mcmle.csv")
df_mcmle$size_data = as.factor(df_mcmle$size_data)
df_mcmle$ratio_noise_data = as.factor(df_mcmle$ratio_noise_data)
df_mcmle_filt = subset(df_mcmle, param_2 <= 6 & param_2 >= 0 & param_1 >= -2 & param_1 <= 6)
```

```
ggplot(df_mcmle_filt, aes(x = param_1, y = param_2, color = size_data, shape = ratio_noise_data)) +
  geom_point(size = 2) +
  scale_color_manual(values = pal2) + labs(shape="Ratio (n/m) : ", col="Taille de l'échantillon (m) : ")
  xlab(expression(mu)) +
  ylab(expression(sigma)) +
  theme(legend.position="bottom", legend.box="vertical", legend.margin=margin()) +
  geom_vline(xintercept = 2, linetype="dotted") +
  geom_hline(yintercept = 4, linetype="dotted") +
  ggtitle("MC MLE : étude du choix des dimensions")
```

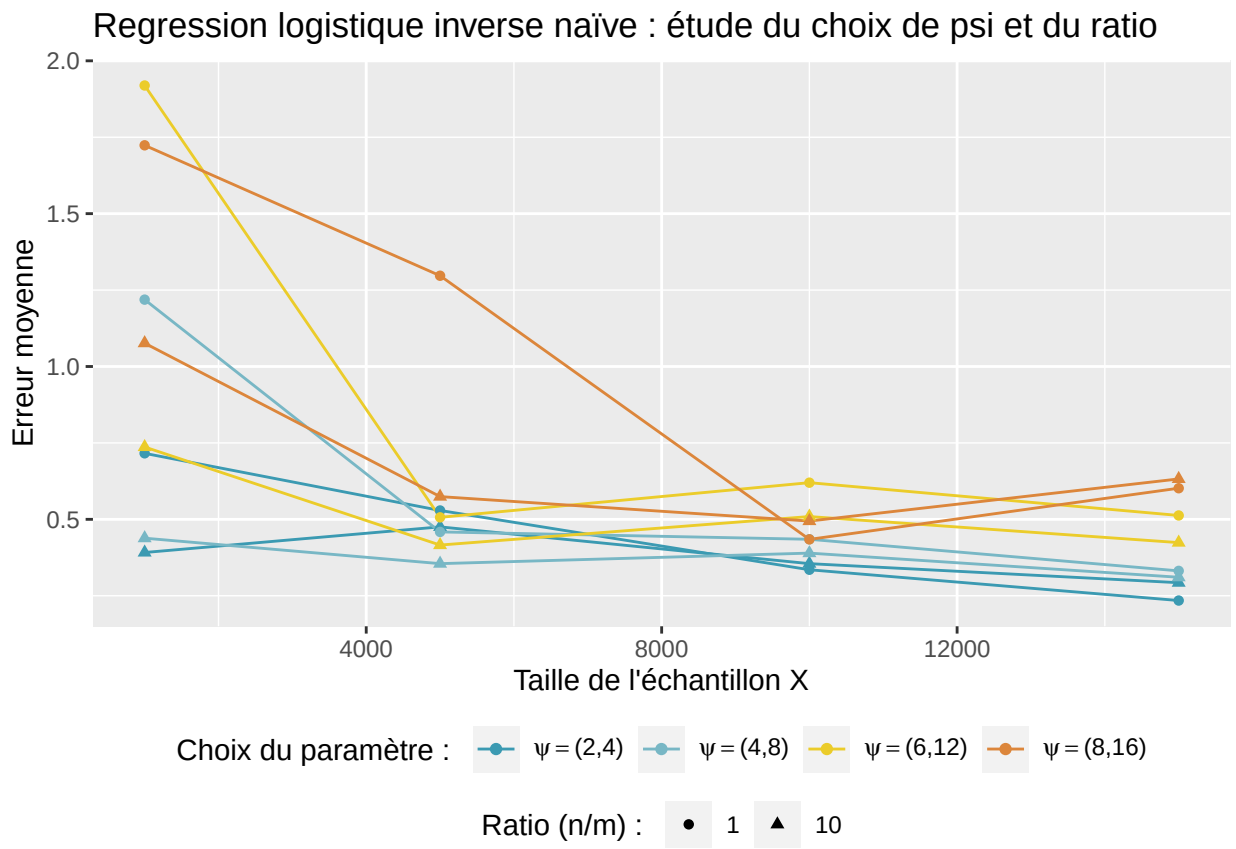



```
#------------- METHODE MC MLE : ETUDE DE PSI ----------------------------------

df_mcmle_psi <- read_csv("dataframes/df_mcmle_psi.csv")
df_mcmle_psi = df_mcmle_psi[order(-df_mcmle_psi$ratio_alpha_psi),]
df_mcmle_psi$ratio_alpha_psi = as.factor(df_mcmle_psi$ratio_alpha_psi)

lab = list(bquote(psi == "(2,4)"), bquote(psi == "(6,12)"), bquote(psi == "(10,20)"), bquote(psi == "(14

ggplot(df_mcmle_psi, aes(x = param_1, y = param_2, color = ratio_alpha_psi)) +
  geom_point(size = 2) +
  scale_color_manual(values = pal4, labels = lab) +
  labs(col=" ") +
  xlab(expression(mu)) +
  ylab(expression(sigma)) +
  theme(legend.position="bottom", legend.box="vertical", legend.margin=margin()) +
  geom_vline(xintercept = 2, linetype="dotted") +
```

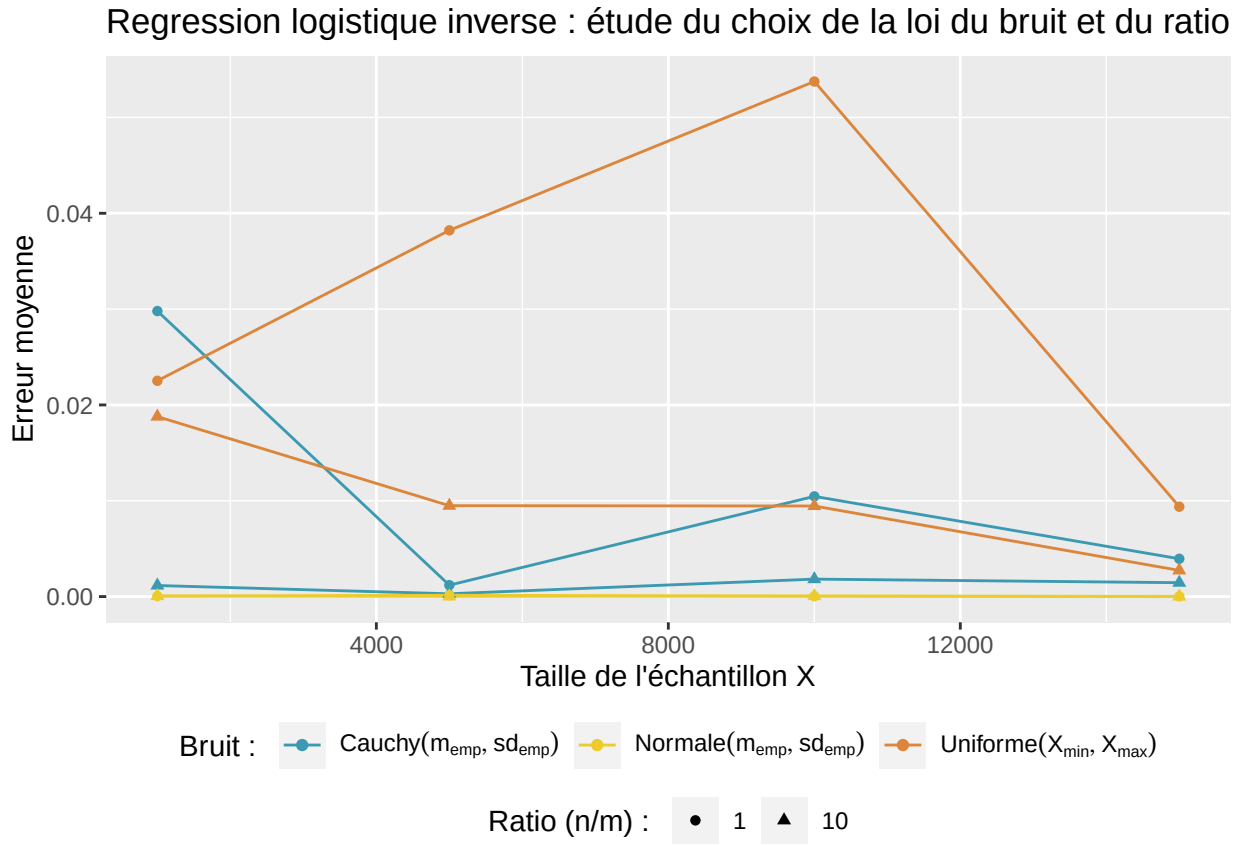
```
  geom_hline(yintercept = 4, linetype="dotted") +
  ggtitle("MC MLE : étude du choix de psi")
```



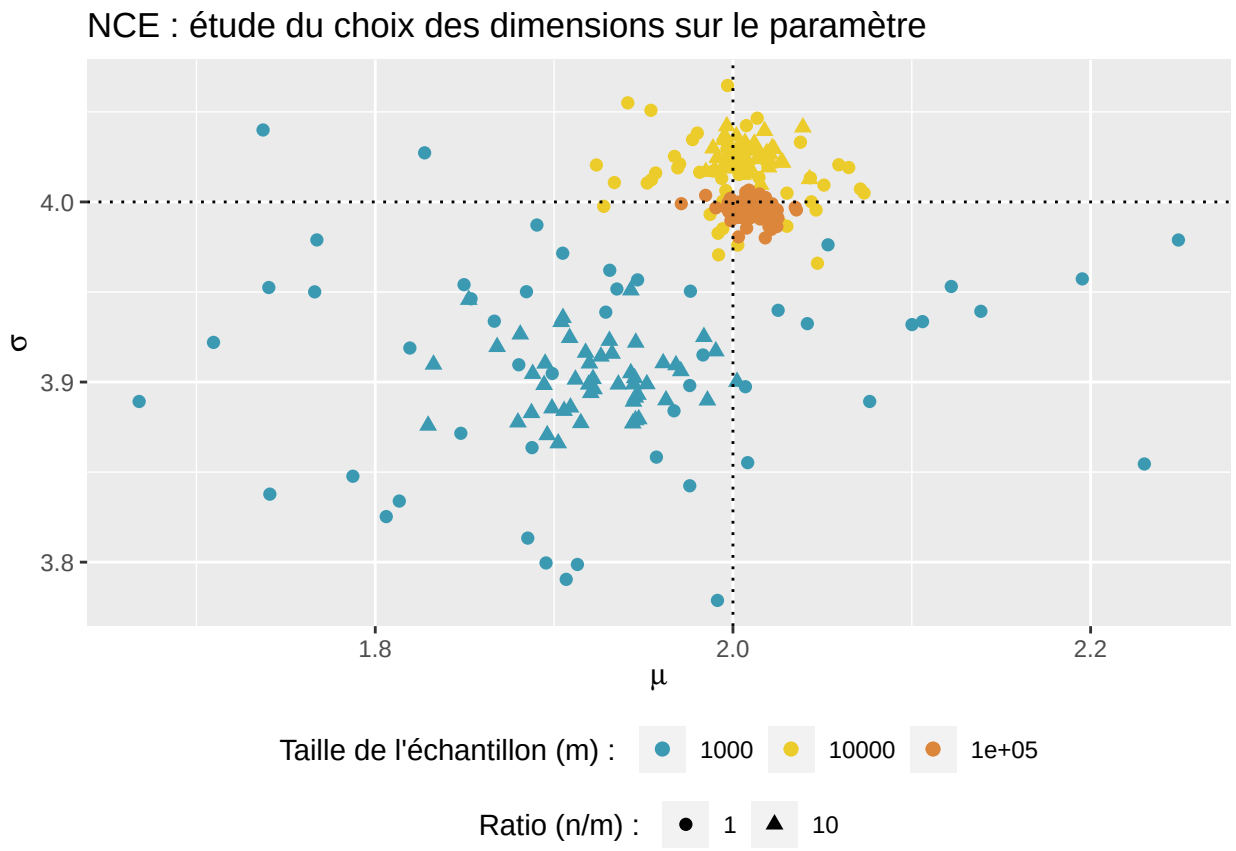

```
#------------- METHODE REGRESSION LOGISTIQUE NAIVE ------------------------------

df_rev_log_reg_naive <- read_csv("dataframes/df_rev_log_reg_naive.csv")
df_rev_log_reg_naive_agg = aggregate(const ~ size_data + ratio_noise_data + ratio_alpha_psi,
                    data = df_rev_log_reg_naive,
                    FUN = mean)
df_rev_log_reg_naive_agg$ratio_alpha_psi = as.factor(df_rev_log_reg_naive_agg$ratio_alpha_psi)
df_rev_log_reg_naive_agg$ratio_noise_data = as.factor(df_rev_log_reg_naive_agg$ratio_noise_data)
df_rev_log_reg_naive_agg$size_data = as.numeric(df_rev_log_reg_naive_agg$size_data)
df_rev_log_reg_naive_agg$const_error = abs(df_rev_log_reg_naive_agg$const - 4*sqrt(2*pi))


ggplot(df_rev_log_reg_naive_agg, aes(x = size_data, y = const_error, color = ratio_alpha_psi, shape = ra
  geom_line() +
  geom_point() +
  labs(shape="Ratio (n/m) : ", col="Choix du paramètre : ") +
  xlab("Taille de l'échantillon X") +
  ylab("Erreur moyenne") +
  theme(legend.position="bottom", legend.box="vertical", legend.margin=margin()) +
  scale_color_manual(values = pal4, labels = list(bquote(psi == "(2,4)"), bquote(psi == "(4,8)"), bquot
  ggtitle("Regression logistique inverse naïve : étude du choix de psi et du ratio")
```



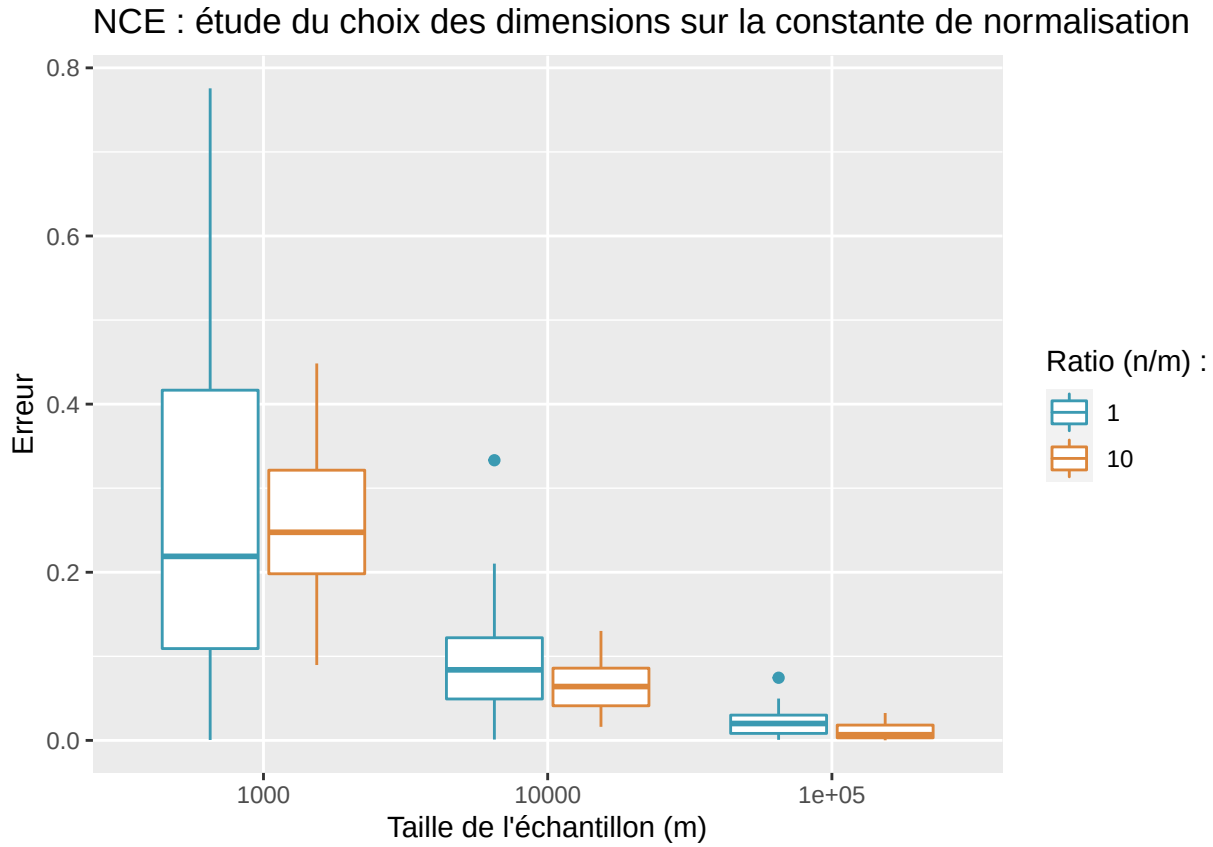

```
#------------- METHODE REGRESSION LOGISTIQUE INVERSE ----------------------------

df_rev_log_reg <- read_csv("dataframes/df_rev_log_reg.csv")
df_rev_log_reg_agg = aggregate(const ~ size_data + ratio_noise_data + law_noise,
                    data = df_rev_log_reg,
                    FUN = mean)
df_rev_log_reg_agg$ratio_noise_data = as.factor(df_rev_log_reg_agg$ratio_noise_data)
df_rev_log_reg_agg$size_data = as.numeric(df_rev_log_reg_agg$size_data)
df_rev_log_reg_agg$const_error = abs(df_rev_log_reg_agg$const - 4*sqrt(2*pi))

ggplot(df_rev_log_reg_agg, aes(x = size_data, y = const_error, color = law_noise, shape = ratio_noise_da
  geom_line() +
  geom_point() +
  scale_color_manual(values = pal3, labels = c(expression(Cauchy(m[emp],sd[emp])), expression(Normale(m
  xlab("Taille de l'échantillon X") +
  ylab("Erreur moyenne") +
  theme(legend.position="bottom", legend.box="vertical", legend.margin=margin()) +
  labs(shape="Ratio (n/m) : ", col="Bruit : ") +
  ggtitle("Regression logistique inverse : étude du choix de la loi du bruit et du ratio")
```



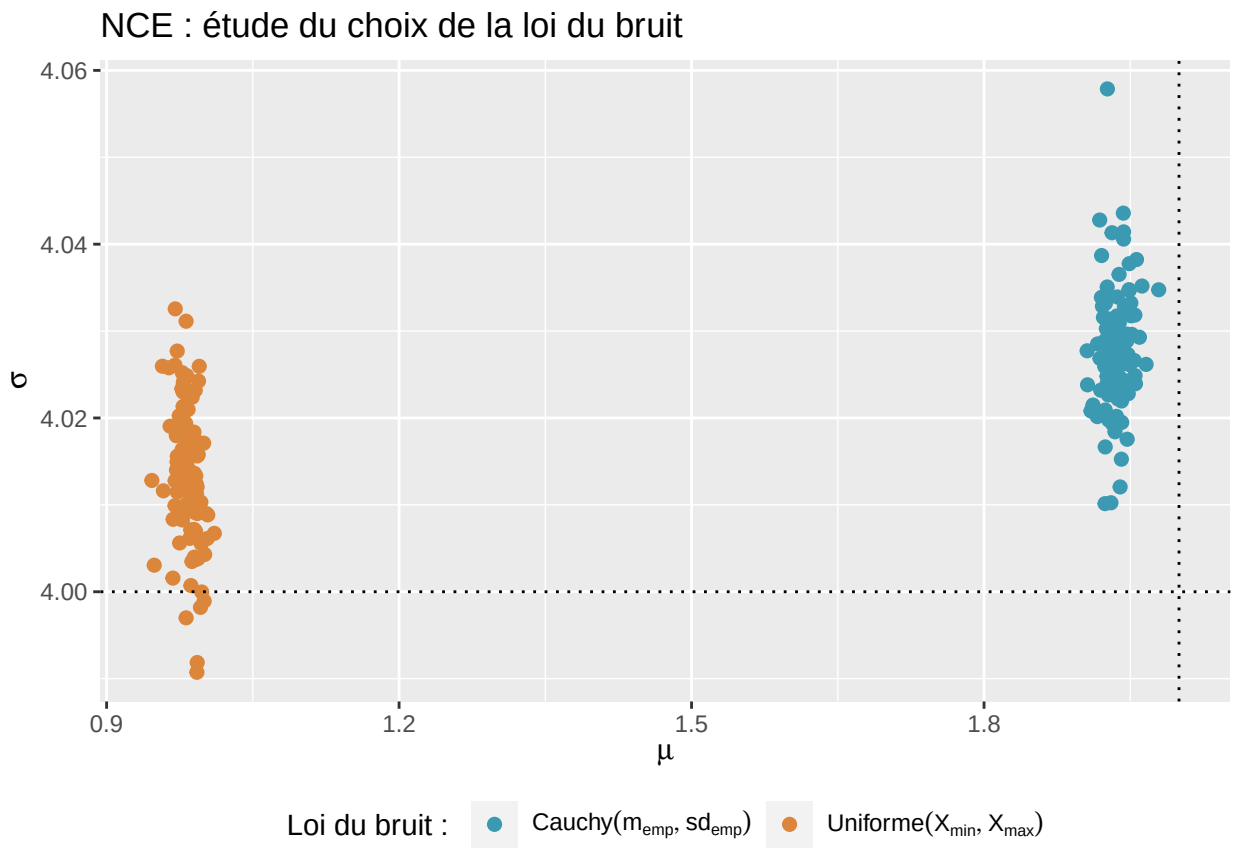

```r
#------------- METHODE NCE : ETUDE DES DIMENSIONS -----------------------------

df_nce <- read_csv("dataframes/df_nce.csv")
df_nce$size_data = as.factor(df_nce$size_data)
df_nce$ratio_noise_data = as.factor(df_nce$ratio_noise_data)
df_nce$const_error = abs(df_nce$const - 4*sqrt(2*pi))

ggplot(df_nce, aes(x = param_1, y = param_2, color = size_data, shape = ratio_noise_data)) +
  geom_point(size = 2) +
  scale_color_manual(values = pal3) +
  labs(shape="Ratio (n/m) : ", col="Taille de l'échantillon (m) : ") +
  xlab(expression(mu)) +
  ylab(expression(sigma)) +
  theme(legend.position="bottom", legend.box="vertical", legend.margin=margin()) +
  geom_vline(xintercept = 2, linetype="dotted") +
  geom_hline(yintercept = 4, linetype="dotted") +
  ggtitle("NCE : étude du choix des dimensions sur le paramètre")
```




```
ggplot(df_nce, aes(x = size_data, y = const_error, color = ratio_noise_data)) + geom_boxplot() + scale_
```




```
#------------- METHODE NCE : ETUDE DES DIMENSIONS -----------------------------

df_nce_noises <- read_csv("dataframes/df_nce_noises.csv")
df_nce_noises$const_error = abs(df_nce_noises$const - 4*sqrt(2*pi))

ggplot(df_nce_noises, aes(x = param_1, y = param_2, color = law)) +
  geom_point(size = 2) +
  scale_color_manual(values = pal2, labels = c(expression(Cauchy(m[emp],sd[emp])), expression(Uniforme(
  labs(col="Loi du bruit : ") +
  xlab(expression(mu)) +
  ylab(expression(sigma)) +
  theme(legend.position="bottom", legend.box="vertical", legend.margin=margin()) +
  geom_vline(xintercept = 2, linetype="dotted") +
  geom_hline(yintercept = 4, linetype="dotted") +
  ggtitle("NCE : étude du choix de la loi du bruit")
```



4.3 Bootstrap

On calcule maintenant les intervalles de confiance des estimateurs de chaque méthode avec du bootstrap. Les données sont tirées des fichiers .csv que nous avons préalablement générés. Les fichiers .csv sont disponibles dans le dossier **/dataframes** du github.

```
'
#------------- CREATION DES CSV ---------------------------------------------------

# Calcul de {size_boot} estimateurs par bootstrap pour la méthode NCE

NCE_bootstrap = function(x, generate_y, psi, log_pm, log_pn, size_theta, n, size_boot) {
  m = length(x)
  theta_bootstrap = c()
  x_bootstrap = x
  for (i in 1:size_boot) {
    theta_bootstrap = append(theta_bootstrap, nce(x_bootstrap,
                                                  generate_y,
                                                  psi,
                                                  log_pm,
                                                  log_pn,
                                                  size_theta,
                                                  n))
    x_bootstrap = sample(x, size = m, replace=TRUE)
  }
```

```
  return(matrix(theta_bootstrap, nrow = size_theta))
}

# Calcul de {size_boot} estimateurs par bootstrap pour la méthode MC MLE

MCMLE_bootstrap = function(x, n, psi, pm_barre, size_boot) {
  m = length(x)
  theta_bootstrap = c()
  x_bootstrap = x
  for (i in 1:size_boot) {
    theta_bootstrap = append(theta_bootstrap, mc_mle(x_bootstrap,
                                                     n,
                                                     psi,
                                                     pm_barre))
    x_bootstrap = sample(x, size = m, replace=TRUE)
  }
  return(matrix(theta_bootstrap, nrow = 2))
}

# Calcul de {size_boot} estimateurs par bootstrap pour la méthode regression logistique inverse

RLI_bootstrap = function(x, generate_y, n, alpha, psi, pm_barre, pn_barre, size_boot) {
  m = length(x)
  theta_bootstrap = c()
  x_bootstrap = x
  for (i in 1:size_boot) {
    theta_bootstrap = append(theta_bootstrap, rev_log_reg(x_bootstrap,
                                                     generate_y,
                                                     n,
                                                     alpha,
                                                     psi,
                                                     pm_barre,
                                                     pn_barre))
    x_bootstrap = sample(x, size = m, replace=TRUE)
  }
  return(matrix(theta_bootstrap, nrow = 1))
}


df_nce_bootstrap = NCE_bootstrap(x, rcauchy, psi, log_pm, log_pn_cauchy, 3, n, 100)
write.csv(as.data.frame(df_nce_bootstrap), "dataframes/df_nce_bootstrap.csv")

df_mcmle_bootstrap = MCMLE_bootstrap(x, n, psi, pm_barre, 100)
write.csv(as.data.frame(df_mcmle_bootstrap), "dataframes/df_mcmle_bootstrap.csv")

df_rli_bootstrap = RLI_bootstrap(x, rcauchy, n, alpha, psi, pm_barre, pn_cauchy, 100)
write.csv(as.data.frame(df_rli_bootstrap), "dataframes/df_rli_bootstrap.csv")
'
```

```
#------------ BOOTSTRAP ------------------------------------------------------

bootstrap = function(matrix, a){
  if(dim(matrix)[1] == 1) {
    df = data.frame(
```

```
      theta = matrix[,1],
      biais = rowMeans(matrix) - matrix[,1],
      IC_inf = rowQuantiles(matrix, probs = c(a/2)),
      IC_sup = rowQuantiles(matrix, probs = c(1-a/2))
    )
    rownames(df) <- NULL
    return(df)
  }
  return(data.frame(
    theta = matrix[,1],
    biais = rowMeans(matrix) - matrix[,1],
    IC = rowQuantiles(matrix, probs = c(a/2, 1-a/2))
  ))
}

a = 0.05

#------------- MC MLE ----------------------------------------------------------

df_mcmle_bootstrap = read.csv("dataframes/df_mcmle_bootstrap.csv")[-1]
df_mcmle_bootstrap = data.matrix(df_mcmle_bootstrap, rownames.force = NA)
kable(bootstrap(df_mcmle_bootstrap, a))
```

theta	biais	IC.2.5.	IC.97.5.
2.067563	-0.0206197	1.838474	2.298756
3.833695	0.1540029	3.863435	4.127140

```
#------------- REGRESSION LOGISTIQUE INVERSE -----------------------------------

df_rli_bootstrap = read.csv("dataframes/df_rli_bootstrap.csv")[-1]
df_rli_bootstrap = data.matrix(df_rli_bootstrap, rownames.force = NA)
kable(bootstrap(df_rli_bootstrap, a))
```

theta	biais	IC_inf	IC_sup
10.0167	0.0115248	9.9981	10.06666

```
#------------- NCE -------------------------------------------------------------

df_nce_bootstrap = read.csv("dataframes/df_nce_bootstrap.csv")[-1]
df_nce_bootstrap = data.matrix(df_nce_bootstrap, rownames.force = NA)
kable(bootstrap(df_nce_bootstrap, a))
```

theta	biais	IC.2.5.	IC.97.5.
1.959010	0.0098534	1.891365	2.039181
4.005404	0.0034242	3.961934	4.062955
10.085015	-0.0129293	9.939171	10.201385

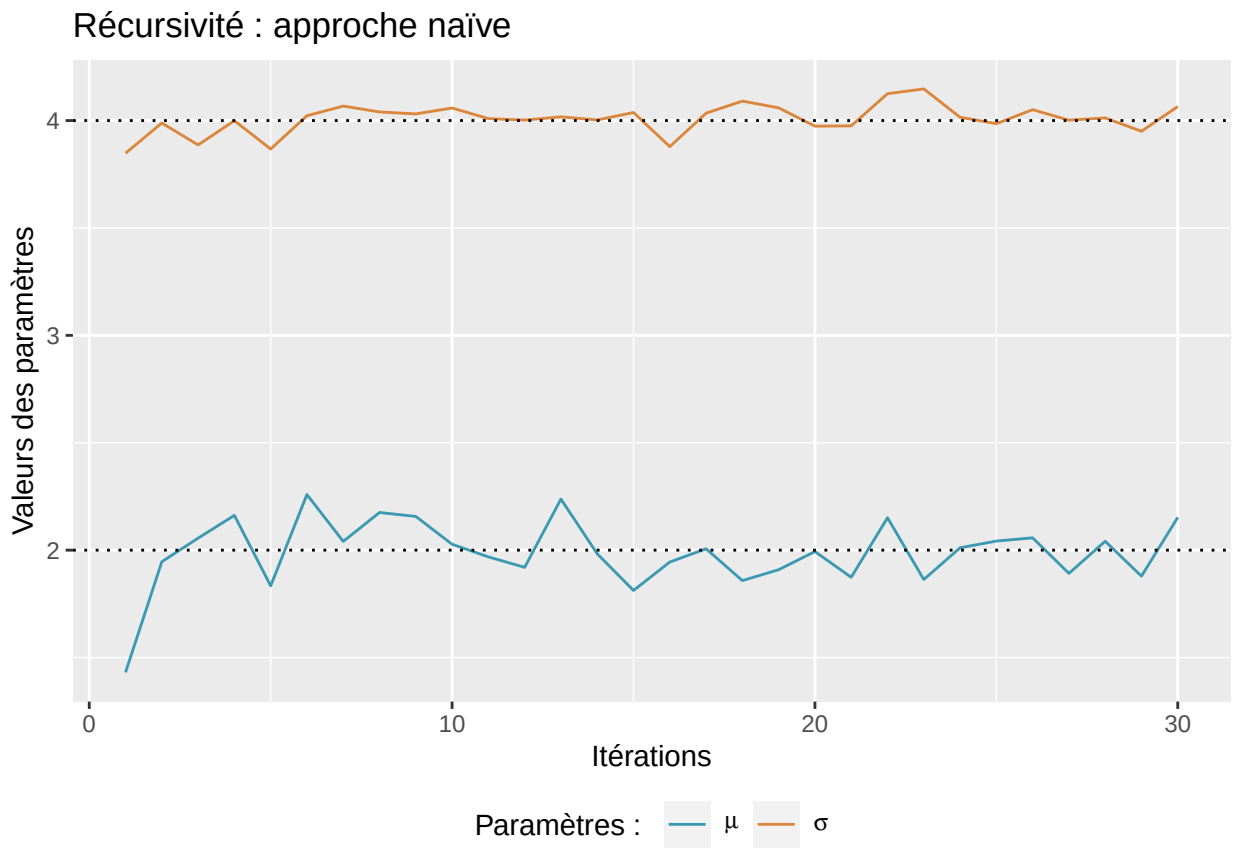
4.4 Approche récursive naïve pour la méthode MC MLE

Idée : améliorer récursivement la précision de la méthode MC MLE en initialisant itération après itération ψ par la précédente estimation de α^*.

Bilan : cette idée naïve n'améliore pas la précision.

```
'
#------------- CREATION DU CSV RECURSIF NAIF ----------------------------------

df_recurs_naive = data.frame(matrix(ncol = 3, nrow = 0))
colnames(df_recurs_naive) = c("iteration", "param_1", "param_2")

psi_init = c(20,20)

for (i in 1:30) {
  psi_init = mc_mle(x, n, psi_init, pm_barre)
  df_recurs_naive[nrow(df_recurs_naive) + 1, ] = c(i, psi_init)
}

write.csv(df_recurs_naive, "dataframes/df_recurs_naive.csv", row.names = FALSE)
'
```

```
#------------- IMPORT DU CSV RECURSIF NAIF ----------------------------------
df_recurs_naive <- read_csv("dataframes/df_recurs_naive.csv")
colnames(df_recurs_naive) = c("iteration", "param_1", "param_2")
df_recurs_naive_melted = melt(df_recurs_naive, id.vars = "iteration")

#------------- GRAPHIQUE ----------------------------------------------------

lab = list(bquote(mu), bquote(sigma))

ggplot(df_recurs_naive_melted, aes(x = iteration, y = value)) +
  geom_line(aes(color = variable, group = variable)) +
  scale_color_manual(values = pal2, labels = lab) +
  labs(col="Paramètres : ") +
  ylab("Valeurs des paramètres") +
  xlab("Itérations") +
  geom_hline(yintercept = 2, linetype="dotted") +
  geom_hline(yintercept = 4, linetype="dotted")  +
  theme(legend.position="bottom", legend.box="vertical", legend.margin=margin()) +
  ggtitle("Récursivité : approche naïve")
```



4.5 Approche récursive avec loi de mélange pour la méthode MC MLE

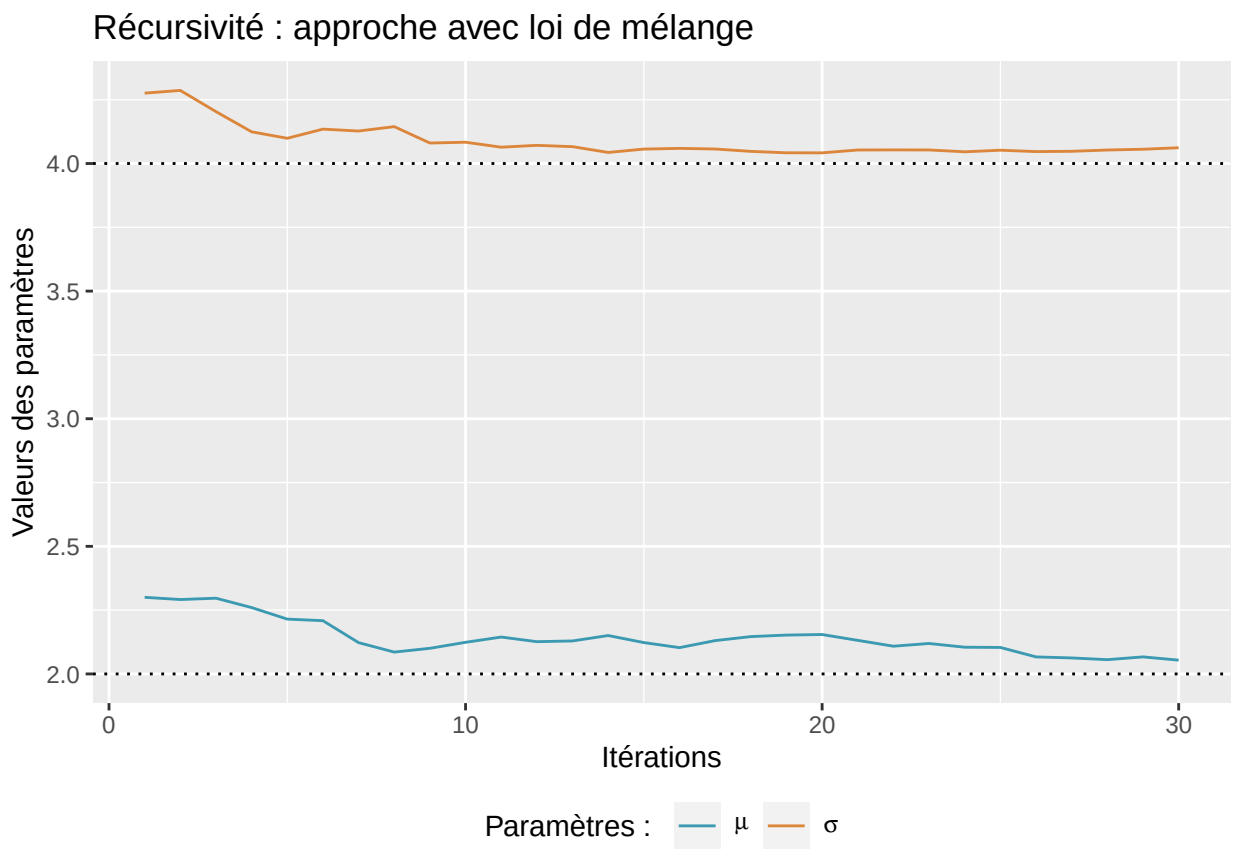
Utilité : améliorer récursivement la précision de la méthode MC MLE en utilisant pour le bruit une loi de mélange des précédentes lois utilisées pour le bruit. Voir la méthode détaillée dans le mémoire.

```
#------------- MCMLE RECURSIF AVEC LOI DE MELANGE -------------------------------

# Entrées : mêmes notations que pour la méthode MCMLE

# Sortie : une matrice comprenant l'ensemble des estimations réalisées au cours des itérations
# la dernière ligne de la matrice correspond au paramètre recherché

mc_mle_recursif = function(x, n, psi, h, iterations){
  m = length(x)

  # la matrice PSI contiendra les estimations de alpha* réalisées au fil des itérations
  # la loi de mélange en dépend intégralement de PSI

  PSI = matrix(ncol = length(psi), nrow = iterations + 1)
  PSI[1, ] = psi

  # initialisation de l\'échantillon de bruit

  Y = c(hasting(x, n, psi, h))

  for (i in 1:iterations) {
```

```
    # définition de la loi de mélange
    # il s\'agit du mélange des lois h(.,psi) pour toutes les valeurs de psi dans le vecteur PSI

    law_mix_h = function(x, PSI){
      size_x = length(x)

      mix = matrix(rep(0, size_x*(i+1)), ncol = size_x, nrow = i+1)

      for (j in 1:i){
        mix[j,] = h(x, PSI[j,])
      }

      return(colSums(mix) / i)
    }

    # définition de la fonction objectif

    L = function(theta){
      return(sum(log(h(x,theta)/law_mix_h(x,PSI))) - m*log(mean(h(Y,theta)/law_mix_h(Y,PSI))))
    }

    # on rajoute au vecteur PSI la dernière estimation réalisée

    PSI[i+1, ] = optim(
      par = rep(1,length(psi)),
      gr = "CG",
      control = list(fnscale=-1),
      fn = L
    )$par

    # on rajoute à l\'échantillon du bruit un nouveau tirage selon la loi h de paramètre PSI[i+1,]

    Y = append(Y, hasting(x, n, PSI[i+1,], h))
  }

  return(PSI)
}

'
#------------- CREATION DU CSV RECURSIF AVEC LOI DE MELANGE --------------------

df_recurs = data.frame(mc_mle_recursif(x, n, c(20,20), pm_barre, 30))[-1,]
df_recurs$iteration = 1:30
colnames(df_recurs) = c("param_1", "param_2","iteration")

write.csv(df_recurs, "dataframes/df_recurs.csv", row.names = FALSE)
'
```

```
#------------- IMPORT DU CSV RECURSIF ------------------------------------------

df_recurs <- read_csv("dataframes/df_recurs.csv")
colnames(df_recurs) = c("param_1", "param_2","iteration")
df_recurs_melted = melt(df_recurs, id.vars = "iteration")
```

```
#------------- GRAPHIQUE ----------------------------------------------------------

lab = list(bquote(mu), bquote(sigma))
ggplot(df_recurs_melted, aes(x = iteration, y = value)) +
  geom_line(aes(color = variable, group = variable)) +
  scale_color_manual(values = pal2, labels = lab) +
  labs(col="Paramètres : ") +
  ylab("Valeurs des paramètres") +
  xlab("Itérations") +
  geom_hline(yintercept = 2, linetype="dotted") +
  geom_hline(yintercept = 4, linetype="dotted") +
  theme(legend.position="bottom", legend.box="vertical", legend.margin=margin()) +
  ggtitle("Récursivité : approche avec loi de mélange")
```



5 Application au modèle d'Ising

5.1 Simulation du modèle d'Ising

Afin de générer des configurations du modèle d'Ising, nous allons dans un premier temps définir des fonctions permettant d'initialiser l'algorithme de Gibbs, d'extraire les voisins d'un site, de calculer l'énergie et la variation de l'énergie.

```
#------------- INITIALISATION --------------------------------------------------

# But : Simule uniformément une matrice de taille height * width composée de -1 et 1
# chaque élément est tiré par une loi uniforme discrète sur {-1,1}

# Sortie : matrice de "height" lignes et "width" colonnes

sim_unif_2D = function(height, width){
  return(matrix(sample(c(-1,1), height * width, replace=T), ncol = width))
}

# But : Simule uniformément un échantillon de n matrices de taille height * width composées de -1 et 1

# Sortie : matrice de "n * height" lignes et "width" colonnes

samples_unif_2D = function(n, height, width){
  return(sim_unif_2D(n * height, width))
}

#------------- VOISINS ---------------------------------------------------------

# But : pour une configuration "config" donnée et un site identifié par son numéro de ligne "row_site"
# et son numéro de colonne "col_site", on cherche les spins voisins

# Sortie : un vecteur avec les spins voisins dans l'ordre HAUT, DROITE, BAS, GAUCHE.
# Le spin vaut 0 s'il n'existe pas

get_neighbors = function(config, row_site, col_site){
  height = dim(config)[1]
  width = dim(config)[2]

  if (row_site == 1){
    up = 0
    down = config[row_site + 1,col_site]
  }
  else if (row_site == height) {
    up = config[row_site - 1,col_site]
    down = 0
  }
  else {
    up = config[row_site - 1,col_site]
    down = config[row_site + 1,col_site]
  }
  if (col_site == 1){
    left = 0
    right = config[row_site, col_site + 1]
  }
```

```
  else if (col_site == width) {
    left = config[row_site, col_site - 1]
    right = 0
  }
  else {
    left = config[row_site, col_site - 1]
    right = config[row_site, col_site + 1]
  }

  return(c(up, right, down, left))
}

#------------ ENERGIE ------------------------------------------------------

# But : calculer l'énergie d'une configuration "config"
# model : argument optionnel utile dans nos tests en dernière section du mémoire

# Sortie : un entier

compute_energy = function(config, model = "ising"){
  height = dim(config)[1]
  width = dim(config)[2]
  energy = 0
  for (i in 1:height){
    for (j in 1:width){
      if (model == "ising") {
        energy = energy - config[i,j]*sum(get_neighbors(config, i, j))
      }
      else {
        # adaptation de l'énergie au modèle de Potts
        # explications de ce besoin dans la dernière section
        energy = energy - sum(config[i,j]==get_neighbors(config, i, j))
      }
    }
  }
  # on divise par deux pour ne pas compter en double les voisins
  return(energy/2)
}


# But : calcule la différence d'énergie entre une config initiale et
# cette même config où le spin à la ligne "row_site" et colonne "col_site" a changé de signe

# Sortie : un entier

compute_delta_energie = function(config, row_site, col_site){
  spin = config[row_site, col_site]
  return(2 * spin * sum(get_neighbors(config, row_site, col_site)))
}
```

Nous implémentons à présent une fonction permettant de simuler une configuration issue du modèle d'Ising à partir de l'algorithme de Gibbs : on part d'une configuration aléatoire. A chaque itération, on sélectionne un site $\sigma_{i,j}$ au hasard. Ce site prend la valeur 1 avec probabilité $p := \mathbb{P}(\sigma_{i,j} = 1|\sigma_{-i,j})$ et -1 avec probabilité $1 - p$. $\sigma_{-i,j}$ est une notation qui désigne les voisins de $\sigma_{i,j}$.

On a $p = \frac{1}{1+exp(\beta \Delta H)}$ avec $\Delta H = 2\sigma_{i,j} \sum_{(k,l) \in -(i,j)} \sigma_{k,l}$.

```
#------------- SIMULATION D'UNE CONFIGURATION ----------------------------------

# But : simuler une configuration du modèle d'ising de paramètre "beta"
#   "iter" est le nombre d'iterations effectuées pour supposément atteindre la stationnarité
#   "init" est une matrice qui représente la configuration initale

# Sortie : une matrice de taille "height" x "width"

sim_ising_2D = function(beta, iter, init) {
  config = init
  height = dim(config)[1]
  width = dim(config)[2]
  for (i in 1:iter) {
    row = sample(1:height, 1)
    col = sample(1:width, 1)
    p = 1/(1 + exp(beta * compute_delta_energie(config, row, col)))
    config[row, col] = sample(c(1,-1), 1, prob = c(p, 1-p))
  }
  return(config)
}
```

Maintenant que l'on sait simuler une configuration issue du modèle d'Ising, on implémente une fonction qui génère un échantillon i.i.d. de configurations. Pour cela on va continuer à faire tourner l'algorithme de Gibbs après avoir atteint la stationnarité et sélectionner les configurations tous les "epsilon" pas.

```
#------------- SIMULATION D'UN ECHANTILLON DE CONFIGURATIONS --------------------

# Entrées :
#   beta : paramètre du modèle
#   height, width : taille d'une configuration
#   burn_in : nombre d'itérations pour atteindre la stationnarité
#   epsilon : pas de décorrélation pour sélectionner des configs iid
#   m : nombre de configurations dans notre échantillon

# Sortie : matrice de hauteur m * height et de largeur width
# autrement dit les configurations sont "empilées" verticalement dans une grande matrice

samples_ising_2D = function(beta, height, width, burn_in, epsilon, m){

  # initialisation
  current_config = sim_unif_2D(height, width)

  # On effectue des itérations jusqu'à atteindre supposément l'état stationnaire
  current_config = sim_ising_2D(beta, burn_in, current_config)
  samples = current_config

  # Une fois la stationnarité atteinte, on prend une configuration tous les epsilon pas
  # afin que les configurations soient iid
  for (i in 2:m){
    current_config = sim_ising_2D(beta, epsilon, current_config)
    samples = rbind(samples, current_config)
  }
```

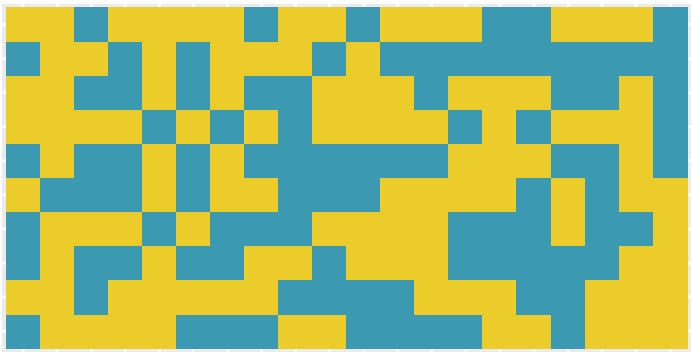
```
  return(samples)
}
```

```
#------------- VISUALISER EN COULEUR UNE CONFIGURATION -------------------------

graph_config = function(config){

  height = dim(config)[1]
  width = dim(config)[2]

  colnames(config) <- paste("Col", 1:width)
  rownames(config) <- paste("Row", 1:height)

  df <- melt(t(config))
  colnames(df) <- c("x", "y", "value")

  return(ggplot(df, aes(x = x, y = y, fill = factor(value)))
         + geom_tile()
         + coord_fixed()
         + scale_fill_manual(values=pal2_bis)
         + labs(col="Spins : ")
         + theme(axis.ticks.x = element_blank(), axis.text.x = element_blank(), axis.ticks.y = element_
         + theme(legend.position = "none")
         )
}

#------------- VISUALISER L'EVOLUTION DE LA CHAINE PENDANT LA PERIODE DE CHAUFFE

beta = 0.5
height = 10
width = 20

# on démarre exceptionnellement d'une matrice composée uniquement de -1
iter0 = matrix(rep(-1, height*width), ncol = width)
graph_config(iter0)
```

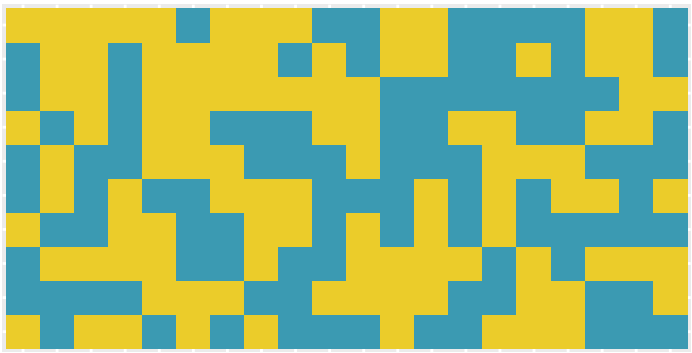
```
iter100 = sim_ising_2D(beta, 100, iter0)
graph_config(iter100)
```

```
iter1000 = sim_ising_2D(beta, 900, iter100)
graph_config(iter1000)
```

```
iter10000 = sim_ising_2D(beta, 9000, iter1000)
graph_config(iter10000)
```



```
iter100000 = sim_ising_2D(beta, 190000, iter10000)
graph_config(iter100000)
```



5.2 Application des méthodes MCMLE, NCE et Regression logistique inverse

```
#------------- DENSITES NORMALISEES ET NON NORMALISEES NECESSAIRES DANS LES METHODES

# Entrées :
#   beta : paramètre du modèle
#   samples : échantillon de plusieurs configurations empilées verticalement dans une matrice
#   model : argument optionnel utile dans nos tests en dernière section du mémoire


# Sortie : densité non normalisée du modèle d'Ising

pm_barre_ising = function(samples, beta, model = "ising"){
  # vecteur qui contiendra l'énergie de chaque configuration
  energy_of_each_config = c()
  # nombre de configurations présentes dans le "samples" en entrée
  m = dim(samples)[1] / height
  for (i in 0:(m-1)){
    # on selectionne une configuration
    id_beginning_config = 1+i*height
    id_end_config = id_beginning_config + height - 1
    # on calcule l'énergie de la configuration sélectionnée
    energy_of_each_config = append(energy_of_each_config, compute_energy(samples[id_beginning_config:id
  }
  return(exp(-beta * energy_of_each_config))
}

# Entrées :
#   theta : paramètres du modèle
#   samples : échantillon de plusieurs configurations empilées verticalement dans une matrice
#   model : argument optionnel utile dans nos tests en dernière section du mémoire

# Sortie : logarithme de la densité du modèle d'Ising

log_pm_ising = function(samples, theta, model = "ising"){
  # theta[1] = beta
  # theta[2] = -log(Z)
  m = dim(samples)[1] / height
  energy_of_each_config = c()
  for (i in 0:(m-1)){
```

```
    id_beginning_config = 1+i*height
    id_end_config = id_beginning_config + height - 1
    energy_of_each_config = append(energy_of_each_config, compute_energy(samples[id_beginning_config:id
  }
  return(-theta[1] * energy_of_each_config + theta[2])
}

# Entrée :
#   samples : échantillon de plusieurs configurations empilées verticalement dans une matrice
#   param : dimensions d'une configuration

# Sortie : densité d'une matrice remplie par une loi uniforme discrète U({-1,1})

pn_discr_unif_2D = function(samples, param){
  # param[1] = height
  # param[2] = width

  # on transforme chaque matrice empilée verticalement dans "samples" en un vecteur
  # ces vecteurs sont empilés verticalement dans une nouvelle matrice "samples"
  samples = matrix(t(samples), ncol = param[1] * param[2], byrow = TRUE)

  # on calcule la densité de chaque "matrice uniforme"
  return(rowProds(ifelse(samples == -1, 1/2, ifelse(samples == 1, 1/2, 0))))
}

log_pn_discr_unif_2D = function(samples, param){
  # param[1] = height
  # param[2] = width
  return(log(pn_discr_unif_2D(samples, c(param[1], param[2]))))
}

#------------- CHOIX DES PARAMETRES ET DIMENSIONS ------------------------------

beta = 0.5
psi = 0.8
width = 20
height = 10
burn_in = 100000
epsilon = 100
m = 100
n = 100
x = samples_ising_2D(beta, height, width, burn_in, epsilon, m)
y_mcmle = samples_ising_2D(psi, height, width, burn_in, epsilon, n)
```

Les fonctions **mc__mle**, **rev__log__reg** et **nce** présentées au début de l'annexe sont valables en dimension de taille 1, c'est à dire lorsqu'un point de donnée est un scalaire et que l'échantillon de données forme un vecteur. Nous allons donc les adapter dans cette section puisque le modèle d'Ising est en dimension supérieure.

```
#------- MC MLE ---------------------------------------------------------------

mc_mle_ising = function(psi,
                        n_ = n,
                        height_ = height,
                        width_ = width,
```

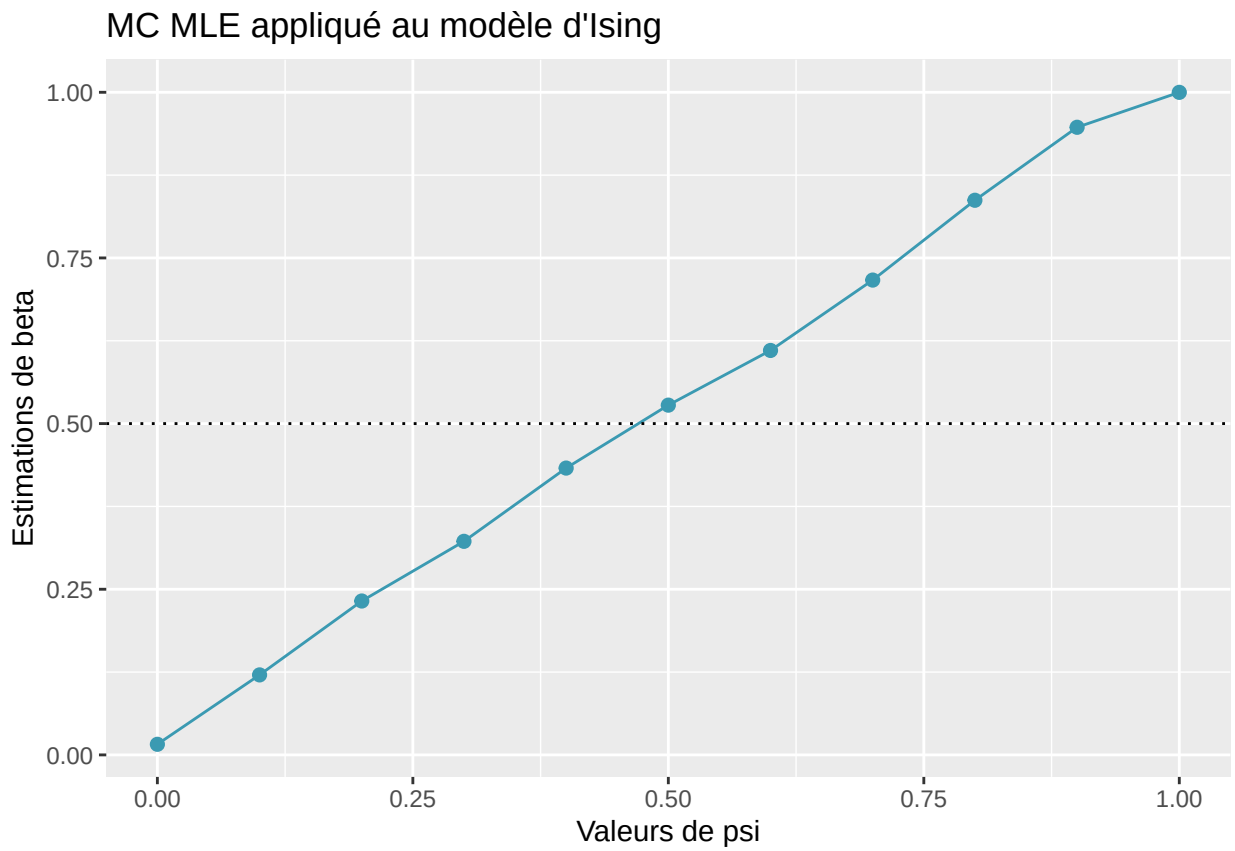
```
                          burn_in_ = burn_in,
                          epsilon_ = epsilon,
                          m_ = m,
                          x_ = x){

  y = samples_ising_2D(psi, height_, width_, burn_in_, epsilon_, n_)

  # Fonction objectif
  L = function(beta){ return(sum(log(pm_barre_ising(x_,beta)/pm_barre_ising(x_,psi))) - m*log(mean(pm_ba

  return(optimize(L, c(0,1), maximum = TRUE)$maximum)
}

(mc_mle_ising(psi))
```

```
## [1] 0.814843
```

Après plusieurs tentatives en faisant varier la valeur de ψ sans toucher à celle de β on remarque que le paramètre estimé semble être ψ au lieu de β.

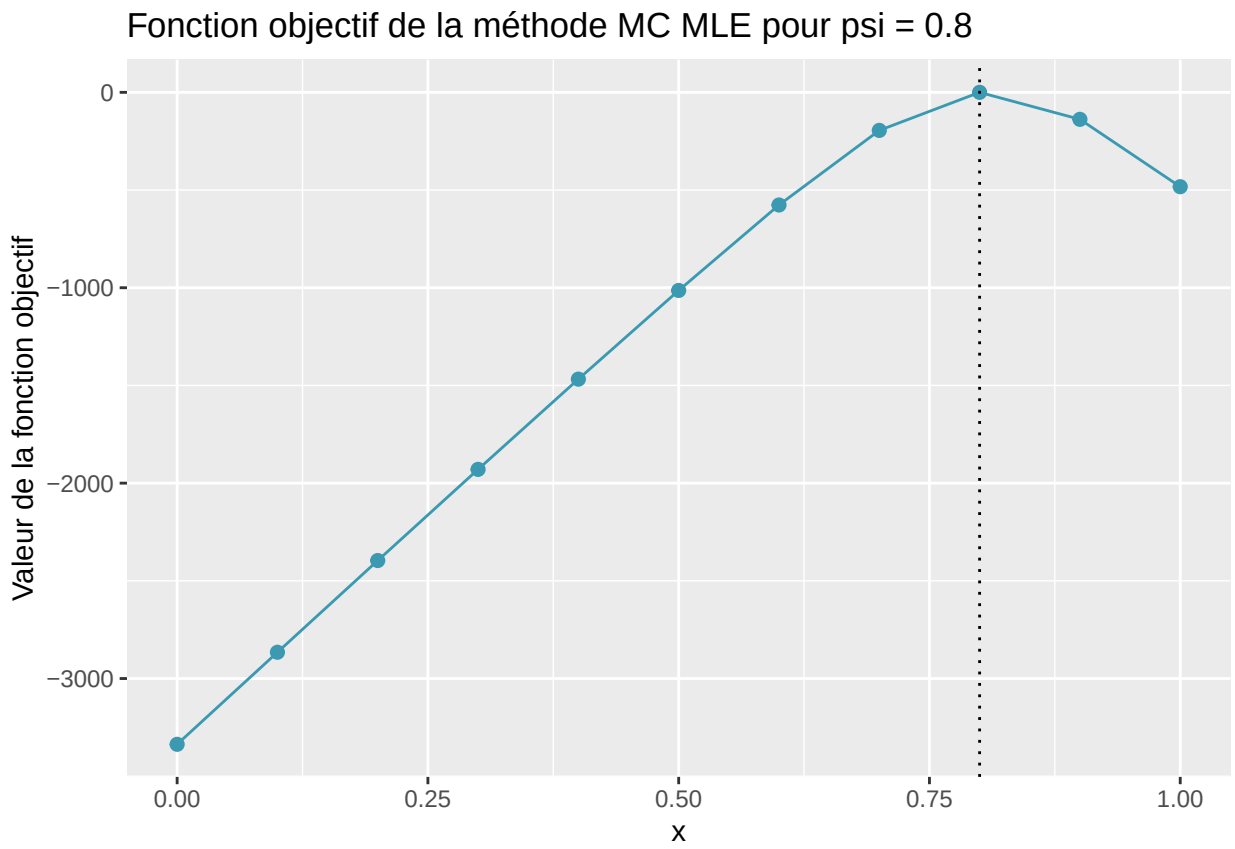
```
'
#------- ESSAI DE DIFFERENTTES VALEURS POUR PSI --------------------------------

PSI = seq(from = 0, to = 1, by = 0.1)
estimations_beta = c()

for (p in PSI){
  estimations_beta = append(estimations_beta, mc_mle_ising(p))
}

df_ising_mc_mle = data.frame(PSI = PSI, estimations_beta = estimations_beta)

write.csv(df_ising_mc_mle, "dataframes/df_ising_mc_mle.csv", row.names = FALSE)

'
```

```
#------- GRAPHIQUE MC MLE ISING ------------------------------------------------

df_ising_mc_mle <- read_csv("dataframes/df_ising_mc_mle.csv")

ggplot(df_ising_mc_mle, aes(x = PSI, y = estimations_beta)) +
  geom_line(color = pal1) +
  geom_point(color = pal1, size = 2) +
  ylab("Estimations de beta") +
  xlab("Valeurs de psi") +
  geom_hline(yintercept = beta, linetype="dotted")  +
  theme(legend.position="bottom", legend.box="vertical", legend.margin=margin()) +
  ggtitle("MC MLE appliqué au modèle d'Ising")
```



Afin de chercher d'où provient l'erreur, on trace la fonction objectif pour la valeur $\psi = 0.8$.

```
#------- GRAPHIQUE FONCTION OBJECTIF MC MLE ISING -----------------------------

abscisse = seq(from = 0, to = 1, by = 0.1)
ordonnee = c()

L = function(beta){ return(sum(log(pm_barre_ising(x,beta)/pm_barre_ising(x,psi))) - m*log(mean(pm_barre_

for (a in abscisse){
  ordonnee = append(ordonnee, L(a))
}

plt = data.frame(x = abscisse, y = ordonnee)

ggplot(plt, aes(x = x, y = y)) +
  geom_line(color = pal1) +
  geom_point(color = pal1, size = 2) +
  ylab("Valeur de la fonction objectif") +
  geom_vline(xintercept = psi, linetype="dotted")  +
  theme(legend.position="bottom", legend.box="vertical", legend.margin=margin()) +
  ggtitle("Fonction objectif de la méthode MC MLE pour psi = 0.8")
```



Nous passons à l'estimation de la constante de normalisation avec les méthode de régression logistique inverse et NCE.

```
#------ REGRESSION LOGISTIQUE INVERSE ----------------------------------------

rli_ising = function (x_ = x,
                      beta_ = beta,
                      height_ = height,
                      width_ = width,
                      m_ = m,
                      n_ = n,
                      model = "ising") {

  # echantillon de bruit
  y = samples_unif_2D(n_, height_, width_)

  # calcul des probabilités p_j
  denom = function(sample, eta) {
    return(pm_barre_ising(sample, beta_, model)*exp(eta[1]) + pn_discr_unif_2D(sample,c(height_, width_
  p_X = function(sample, eta){
    return (pm_barre_ising(sample, beta_, model)*exp(eta[1]) / denom(sample, eta))}
  p_Y = function(sample, eta){
    return (pn_discr_unif_2D(sample, c(height_, width_))*exp(eta[2]) / denom(sample, eta))}

  # fonction objectif
  L = function(eta) {
    return(sum(log(p_X(x_, eta))) + sum(log(p_Y(y, eta))))}
```

```
  # initialisation descente de gradient
  eta1_init = log(m_/(m_+n_))
  eta2_init = log(n_/(m_+n_))

  # optimisation
  const = optim(
    par = c(eta1_init,eta2_init),
    gr = "CG",
    control = list(fnscale=-1),
    fn = L
  )$par

  # calcul de la constante
  constante_additive = const[2] + log(1) - log(n_/(m_+n_))

  # affichage de la constante de normalisation
  return(exp(-const[1] + log(m_/(m_+n_)) + constante_additive))
}

#------ Application avec les paramètres par défaut -----------------------------

(rli_ising())
```

```
## [1] 8.701347e+59
```

```
# ------- NCE ---------------------------------------------------------------

nce_ising = function(x_ = x,
                     height_ = height,
                     width_ = width,
                     n_ = n,
                     m_ = m) {

  # echantillon de bruit
  y = samples_unif_2D(n_, height_, width_)

  # fonction objectif
  h = function(u, theta){
      return( 1 / (1 + n_/m_ * exp(log_pn_discr_unif_2D(u, c(height_, width_)) - log_pm_ising(u, theta))
    }

  J = function(theta){
    return( sum(log(h(x_, theta))) + sum(log(1 - h(y, theta))) )
  }

  # estimation du paramètre et de la constante
  theta = optim(
    par = c(0,-138),
    gr = "CG",
    control = list(fnscale=-1),
    fn = J
  )$par

  return(c(theta[1], exp(-theta[2])))
```

```
}

(nce_ising())
```

```
## [1] 1.847062e-03 1.604284e+60
```

Afin de vérifier nos résultats, nous pouvons utiliser le package GiRaf. Celui-ci est cependant conçu pour le modèle de Potts, dans lequel l'énergie n'est pas définie comme dans le modèle d'Ising. Afin de comparer des quantités comparables, nous avons modifié la fonction compute_energy afin d'y ajouter un argument optionnel qui l'adapte au modèle de Potts

```
# ------- CONSTANTE CALCULEE PAR LE PACKAGE GIRAF -----------------------------
NC.mrf(height, width, beta)
```

```
## [1] 4.455507e+105
```

```
# ------- CONSTANTE ESTIMEE PAR REGRESSION LOGISTIQUE INVERSE -----------------
(rli_ising(model = "Potts"))
```

```
## [1] 2.258661e+100
```